Jigsaw:
Scalable Software-Defined Caches

Nathan Beckmann and Daniel Sanchez
MIT CSAIL

PACT'13 - Edinburgh, Scotland
Sep 11, 2013

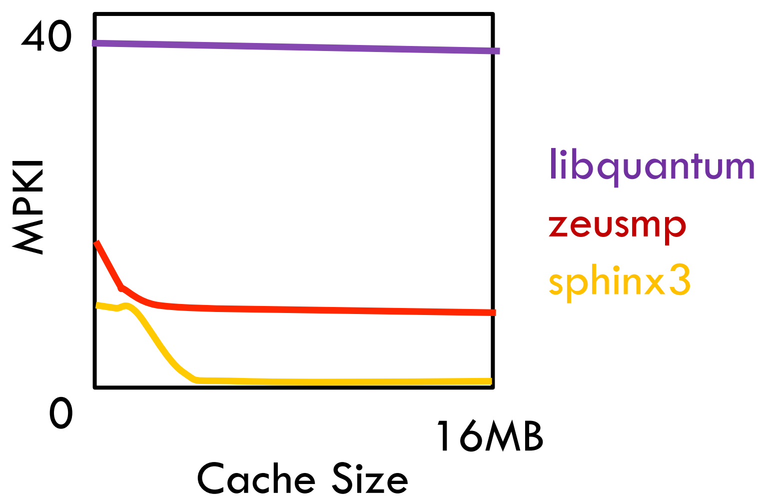
Massachusetts Institute of Technology

CSAIL

Summary

- NUCA is giving us more capacity, but further away

- Applications have widely varying cache behavior

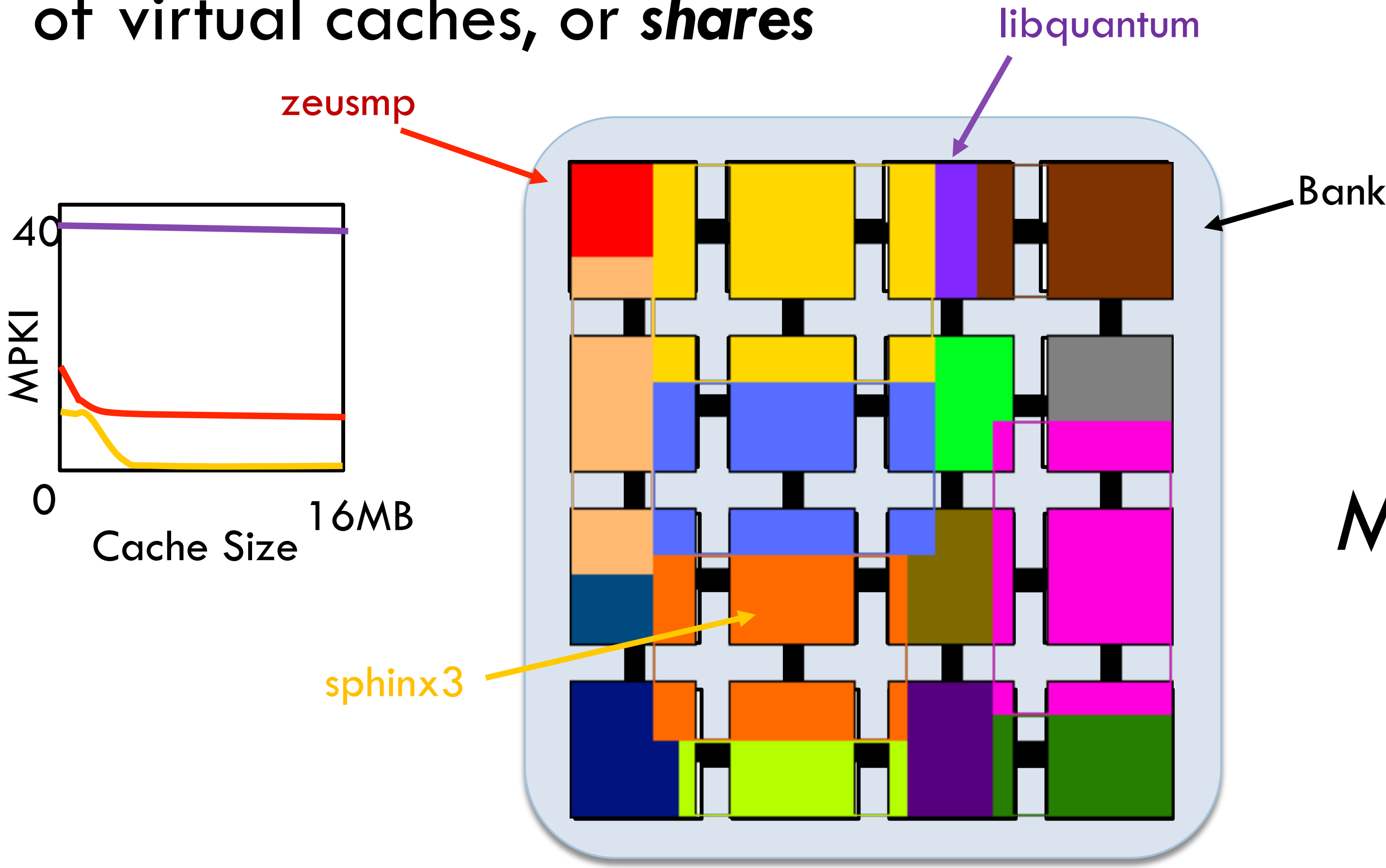


- Cache organization should adapt to application

- Jigsaw uses physical cache resources as building blocks of virtual caches, or ***shares***

Approach

- Jigsaw uses physical cache resources as building blocks of virtual caches, or ***shares***



Tiled Multicore

Agenda

- Introduction
- Background
  - Goals
  - Existing Approaches
- Jigsaw Design
- Evaluation

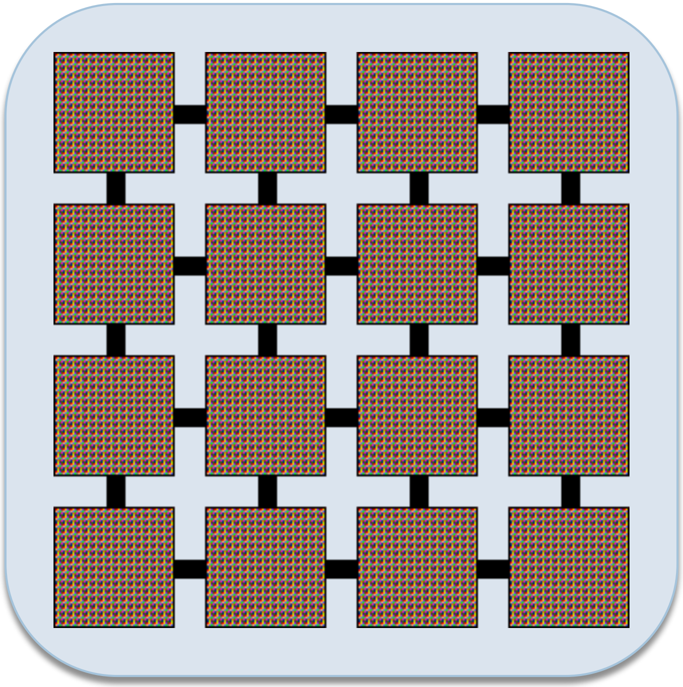
Goals

- Make effective use of cache capacity
- Place data for low latency
- Provide capacity isolation for performance
- Have a simple implementation

Existing Approaches: S-NUCA

Spread lines evenly across banks

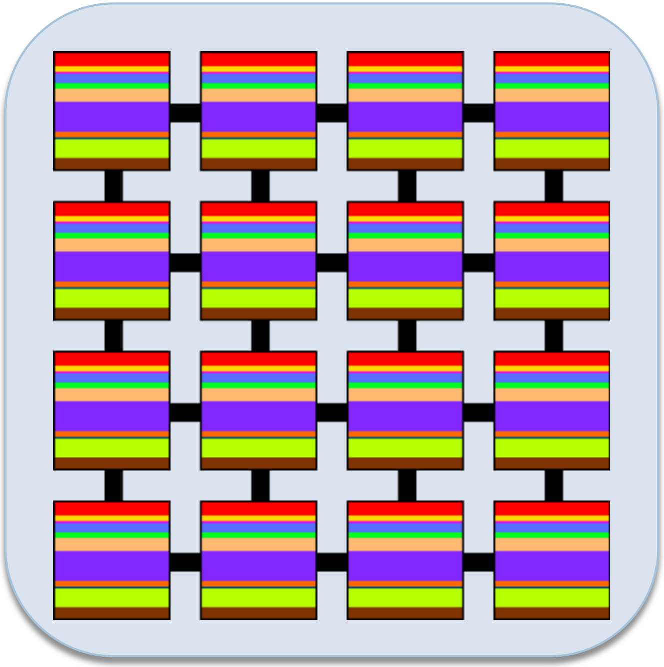
- High Capacity
- High Latency
- No Isolation
- Simple



Existing Approaches: Partitioning

Isolate regions of cache between applications.

- High Capacity
- High Latency
- Isolation
- Simple



- Jigsaw needs partitioning; uses Vantage to get strong guarantees with no loss in associativity

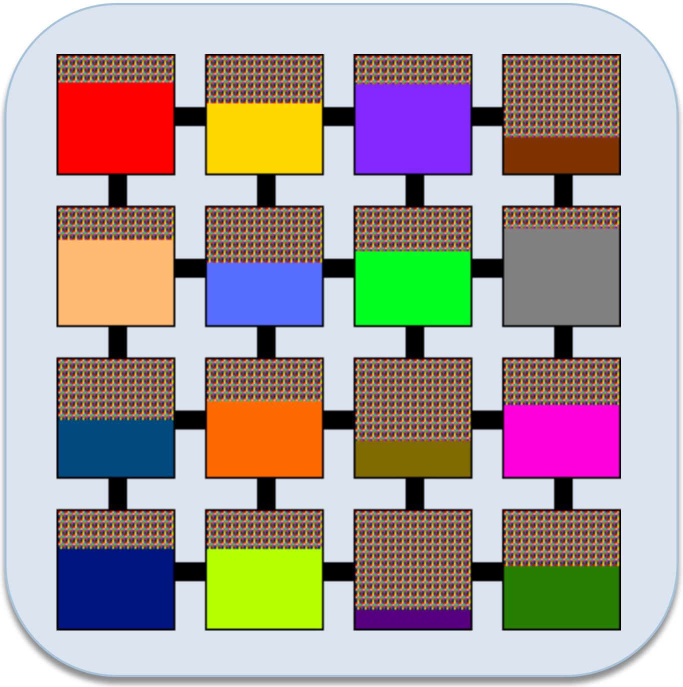
Existing Approaches: Private

Place lines in local bank

- Low Capacity
- Low Latency
- Isolation
- Complex – LLC directory

Existing Approaches: D-NUCA

Placement, migration, and replication heuristics



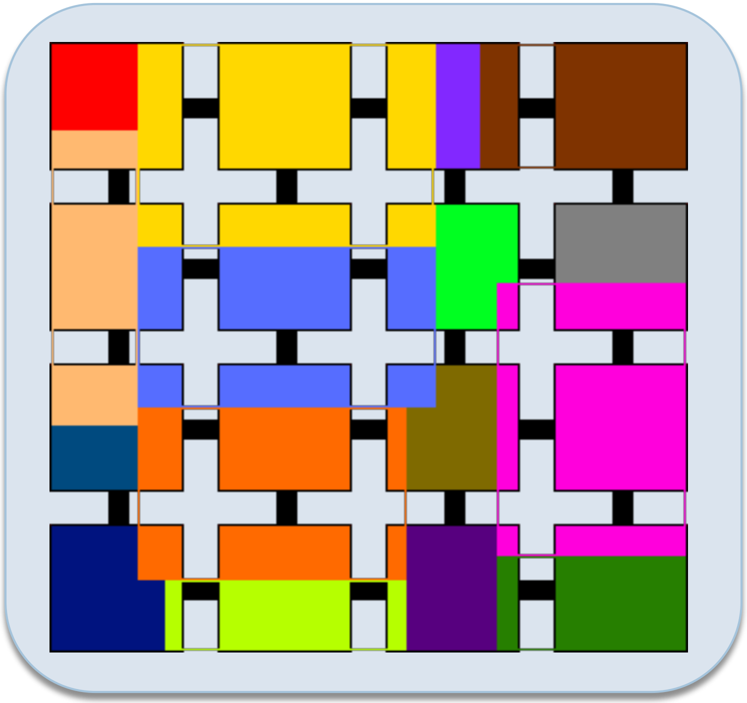
- High Capacity
  - But beware of over-replication and restrictive mappings
- Low Latency
  - Don't fully exploit capacity vs. latency tradeoff
- No Isolation
- Complexity Varies
  - Private-baseline schemes require LLC directory

Existing Approaches: Summary

	S-NUCA	Partitioning	Private	D-NUCA
High Capacity	Yes	Yes	No	Yes
Low Latency	No	No	Yes	Yes
Isolation	No	Yes	Yes	No
Simple	Yes	Yes	No	Depends

Jigsaw

- High Capacity – Any share can take full capacity, *no replication*
- Low Latency – Shares allocated near cores that use them
- Isolation – Partitions within each bank

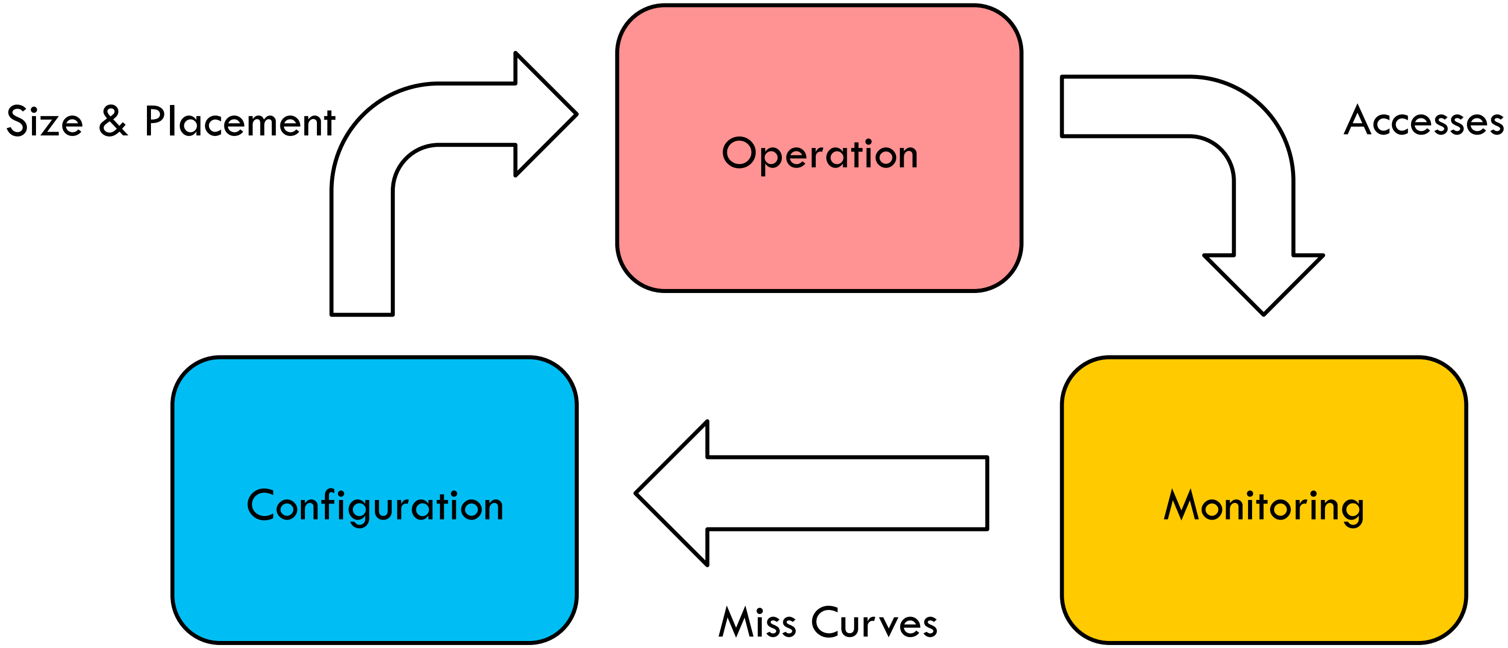


- Simple – Low overhead hardware, no LLC directory, software-managed

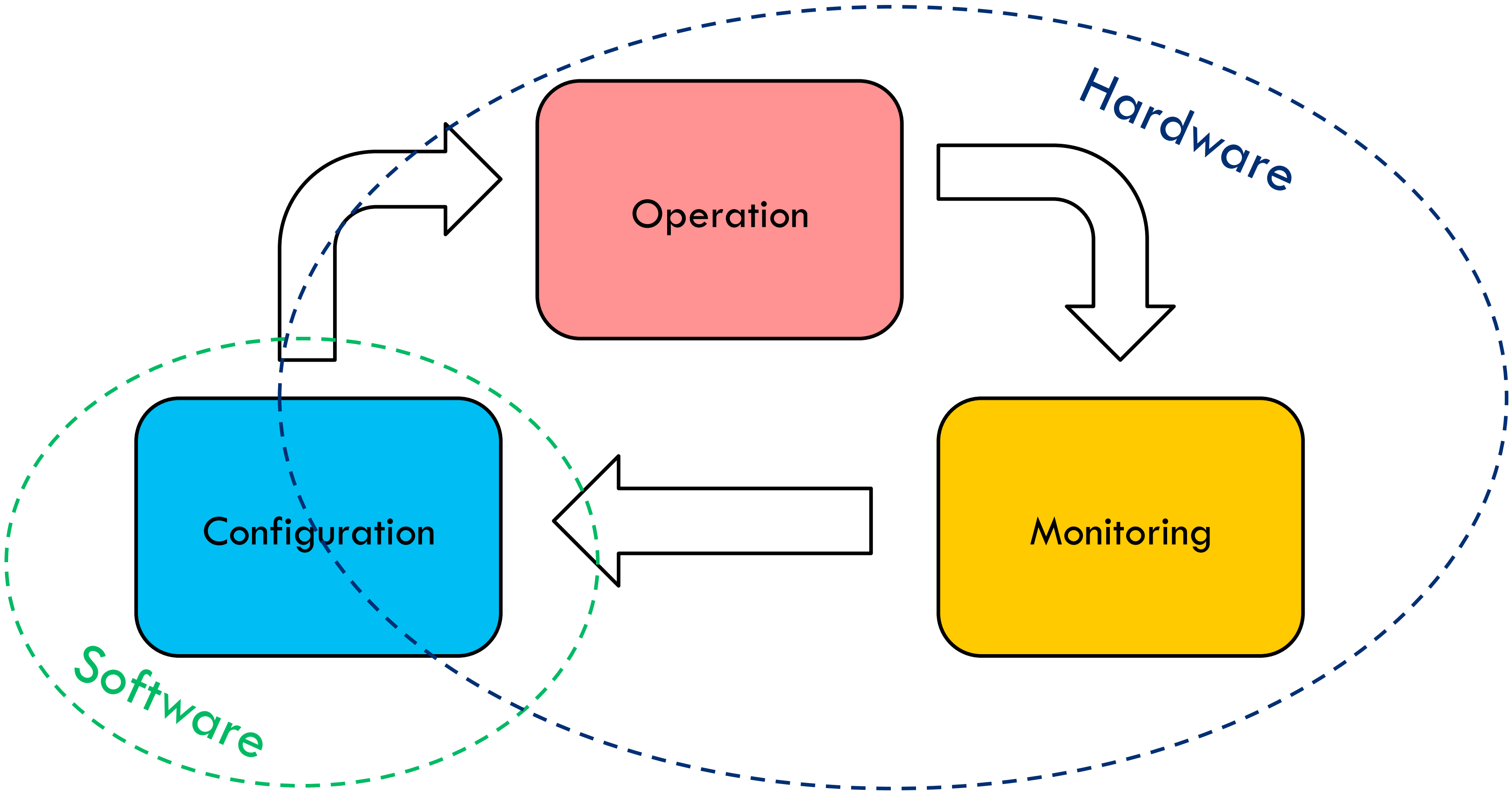
Agenda

- Introduction
- Background
- Jigsaw Design
  - Operation
  - Monitoring
  - Configuration
- Evaluation

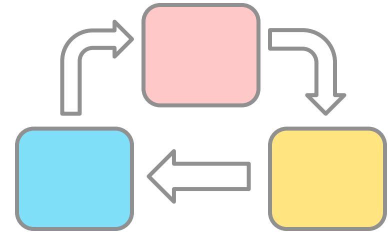
Jigsaw Components



Jigsaw Components



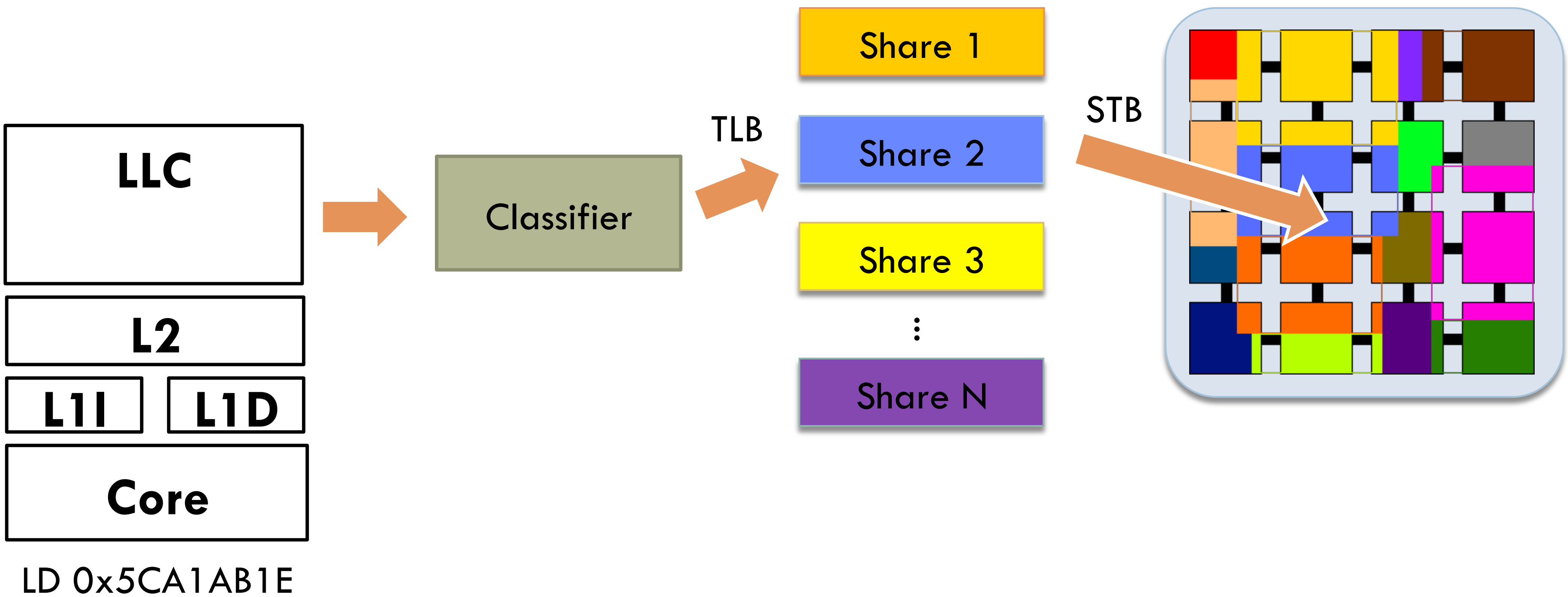
Agenda



- Introduction
- Background
- Jigsaw Design
  - Operation
  - Monitoring
  - Configuration
- Evaluation

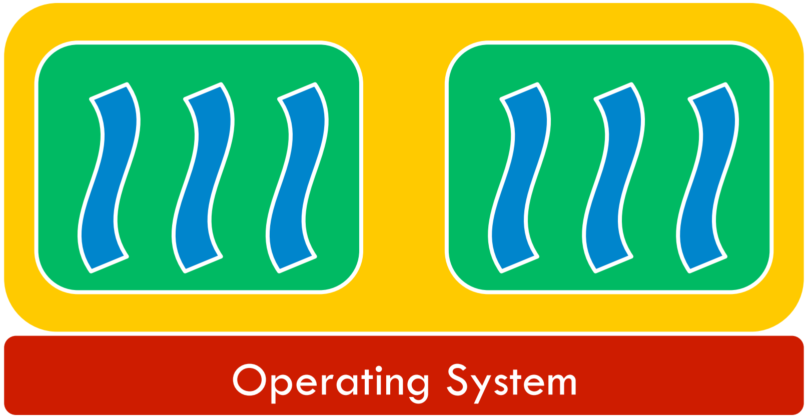
Operation: Access

Data → shares, so **no LLC coherence required**



Data Classification

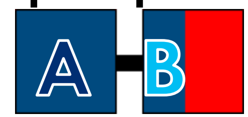
- Jigsaw classifies data based on access pattern
  - Thread, Process, Global, and Kernel



- 6 thread shares
- 2 process shares
- 1 global share
- 1 kernel share

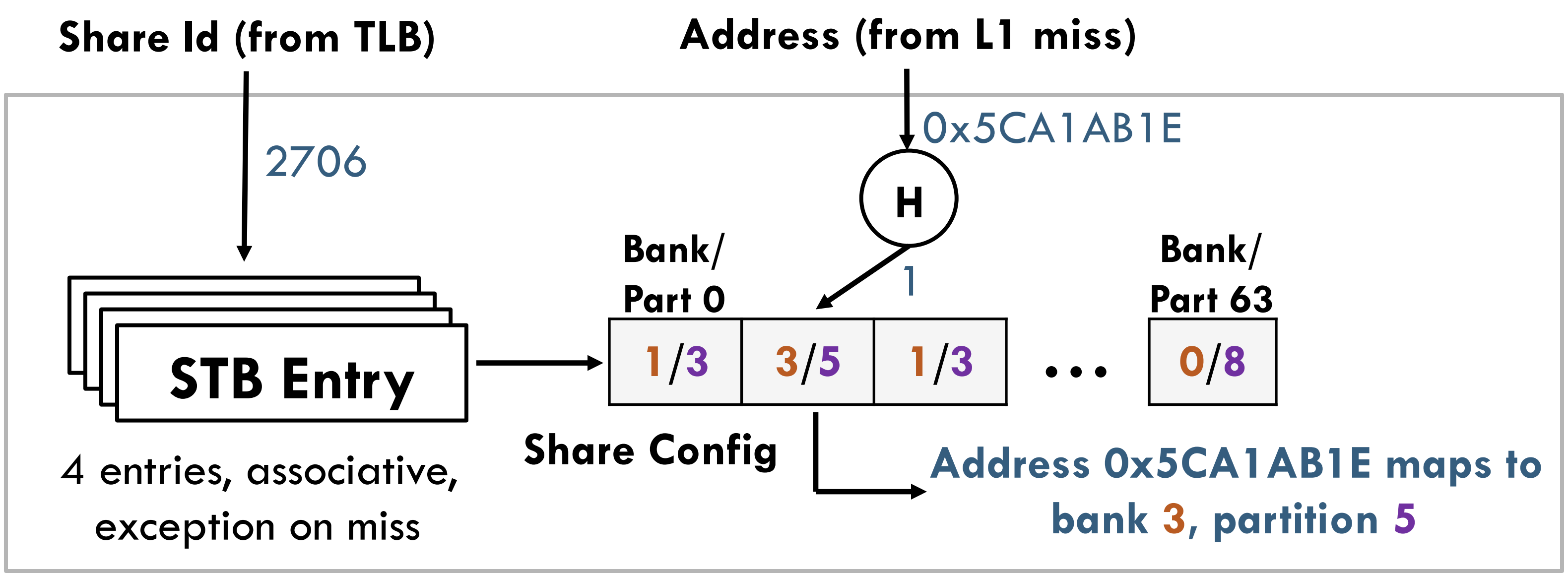
- Data lazily re-classified on TLB miss
  - Similar to R-NUCA but…
    - R-NUCA: Classification → Location
    - Jigsaw: Classification → Share (sized & placed dynamically)
  - Negligible overhead

Operation: Share-bank Translation Buffer

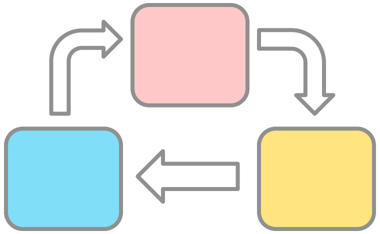
- Gives unique location of the line in the LLC
- Address, Share ➔ Bank, Partition
- Hash lines proportionally
  - Share: 

  - STB: 

- 400 bytes; low overhead



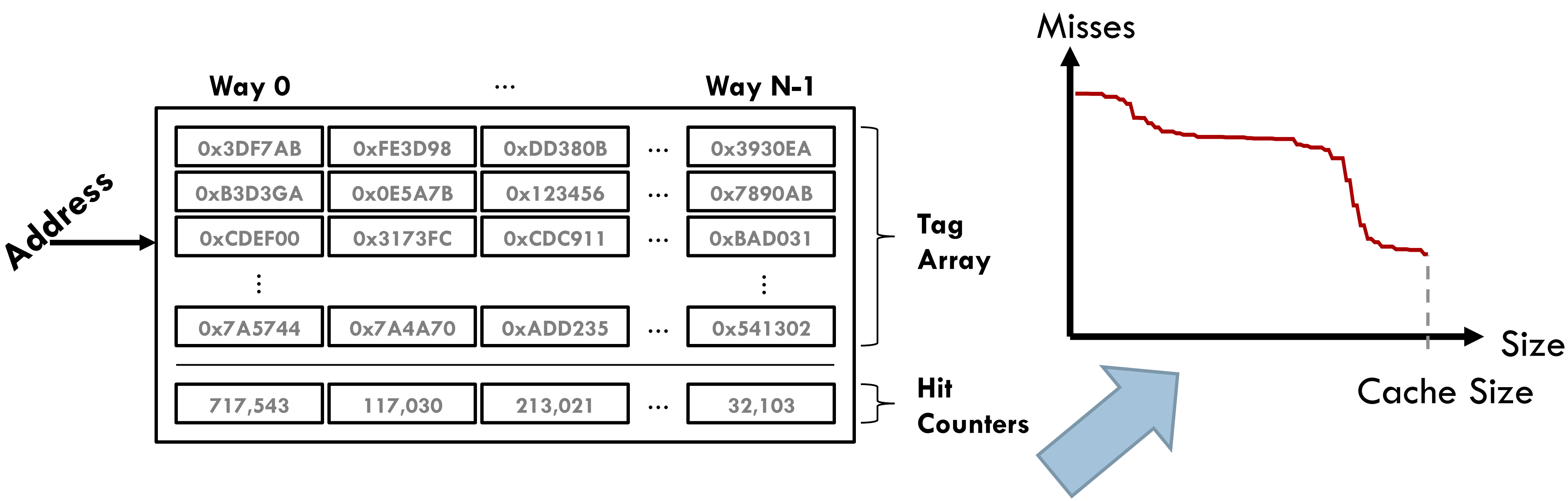
Agenda



- Introduction
- Background
- Jigsaw Design
  - Operation
  - Monitoring
  - Configuration
- Evaluation

Monitoring

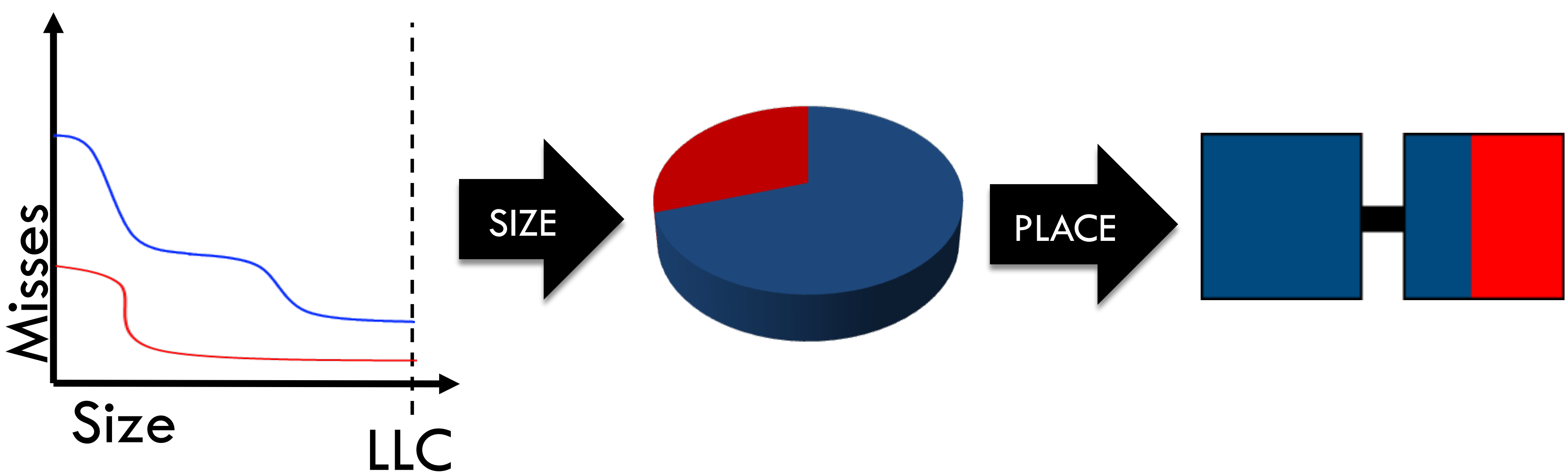
- Software requires miss curves for each share
- Add utility monitors (UMONs) per tile to produce miss curves
- Dynamic sampling to model full LLC at each bank; see paper



Configuration

- Software decides share configuration

- Approach: Size → Place
  - Solving independently is simple
  - Sizing is hard, placing is easy



Configuration: Sizing

- Partitioning problem: Divide cache capacity of **S** among **P** partitions/shares to maximize hits

- Use miss curves to describe partition behavior

- NP-complete in general

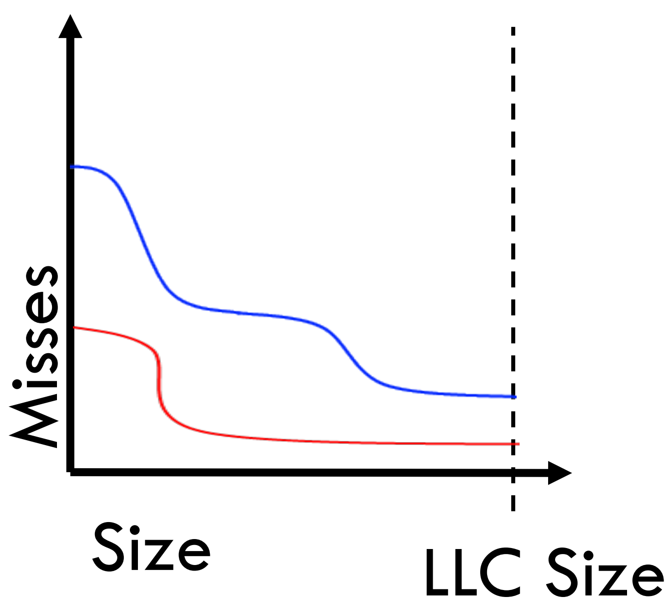
- Existing approaches:
  - ~~Hill climbing is fast but gets stuck in local optima~~
  - UCP Lookahead is good but scales quadratically: $O(P \times S^2)$

    Utility-based Cache Partitioning, Qureshi and Patt, MICRO'06

Can we scale Lookahead?

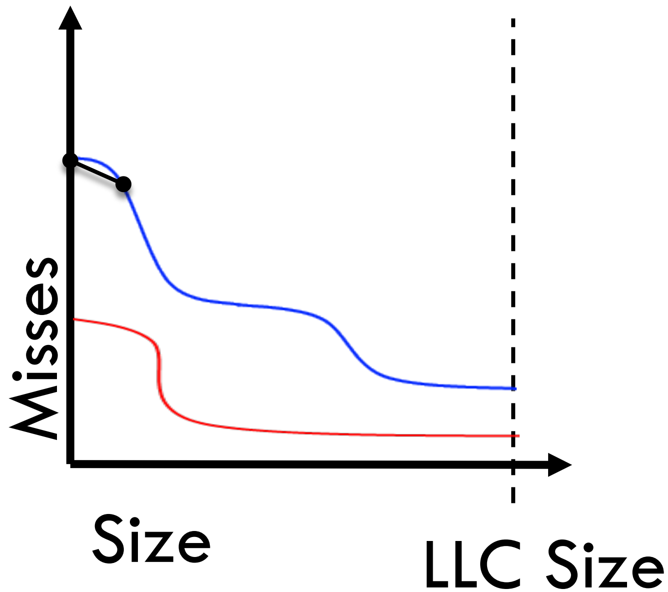
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



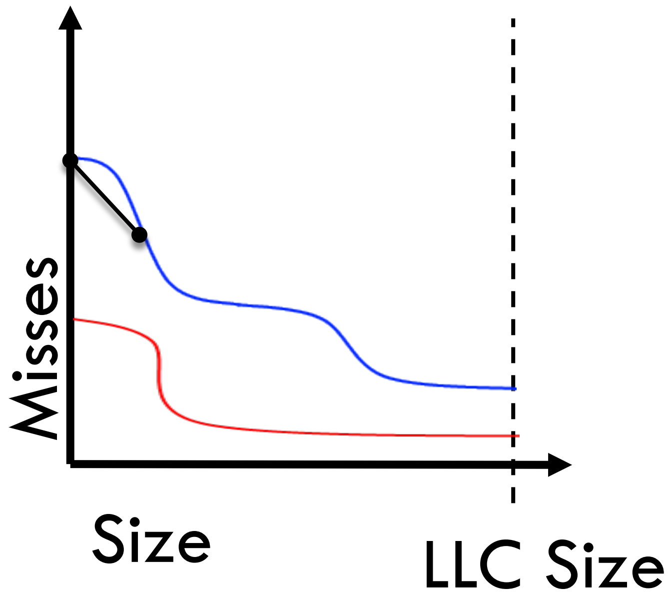
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



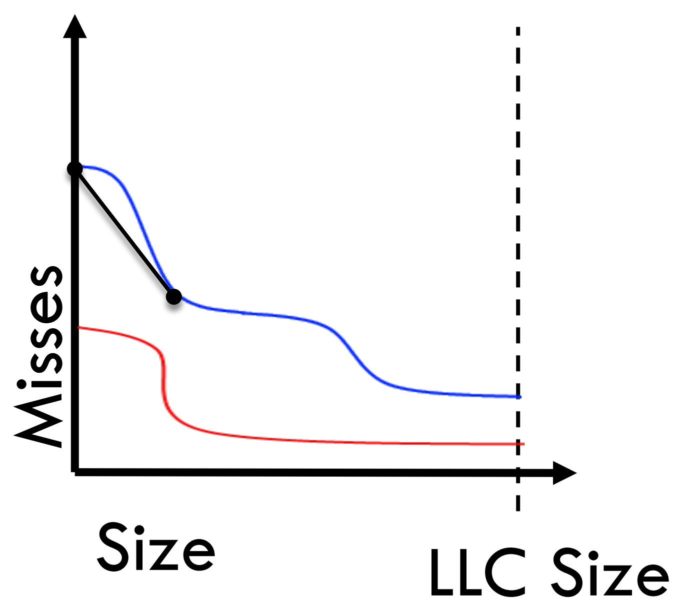
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



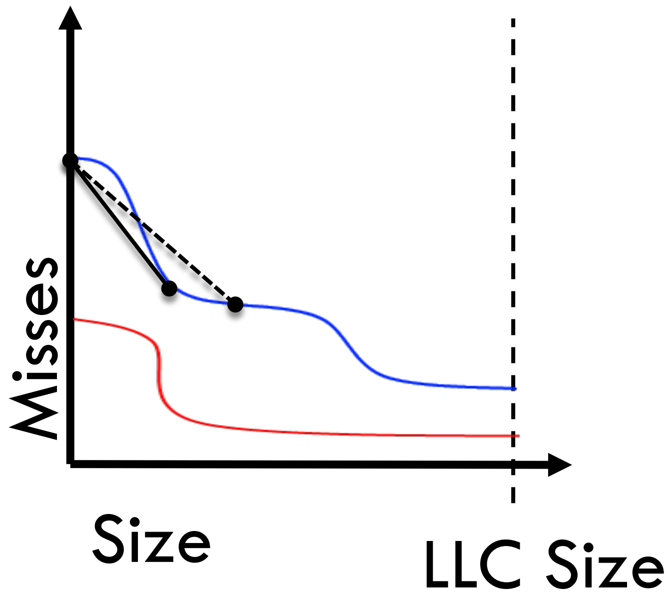
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



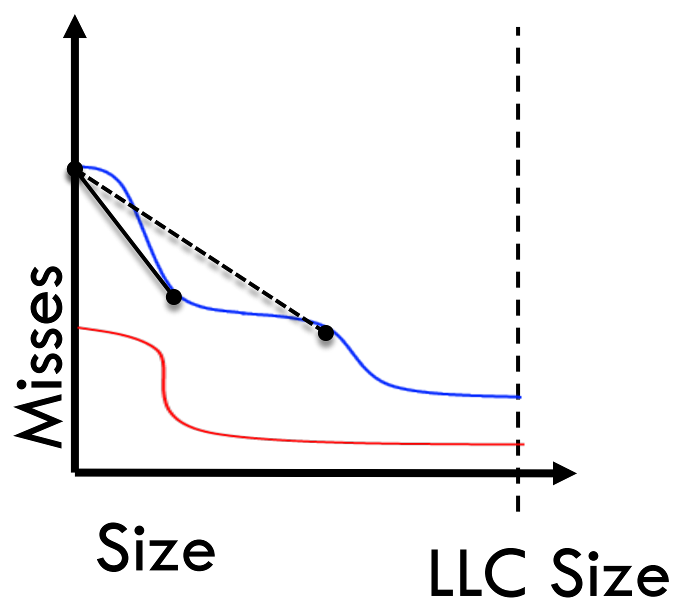
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



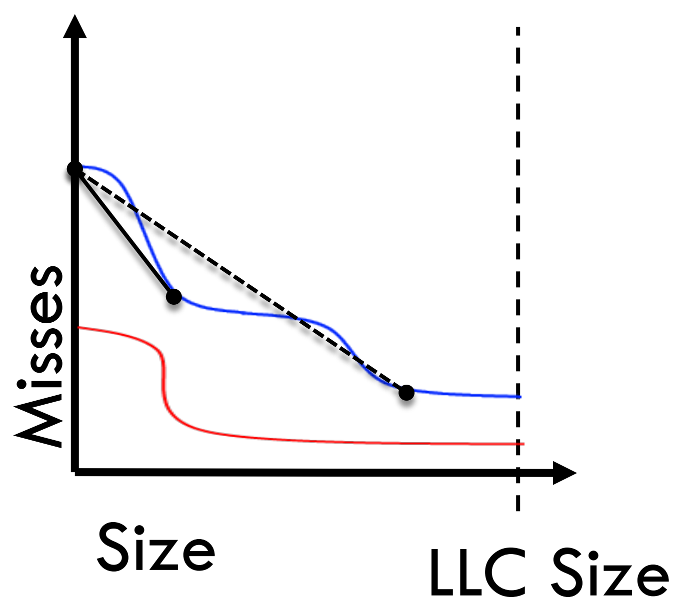
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



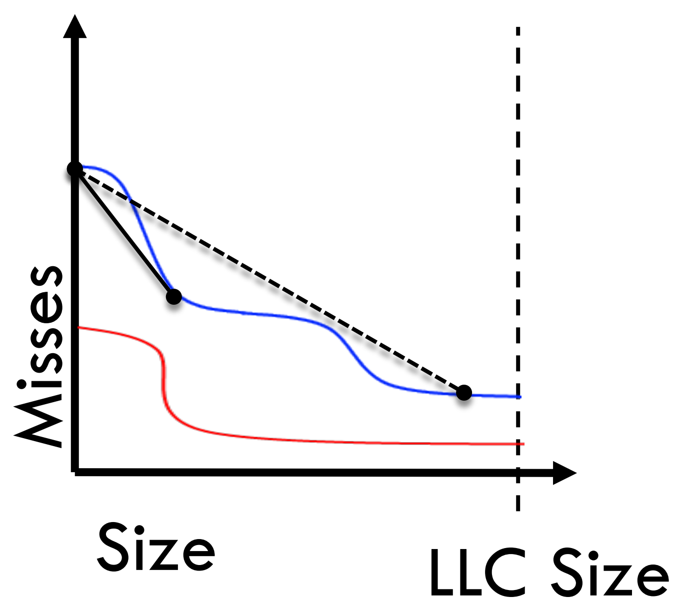
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



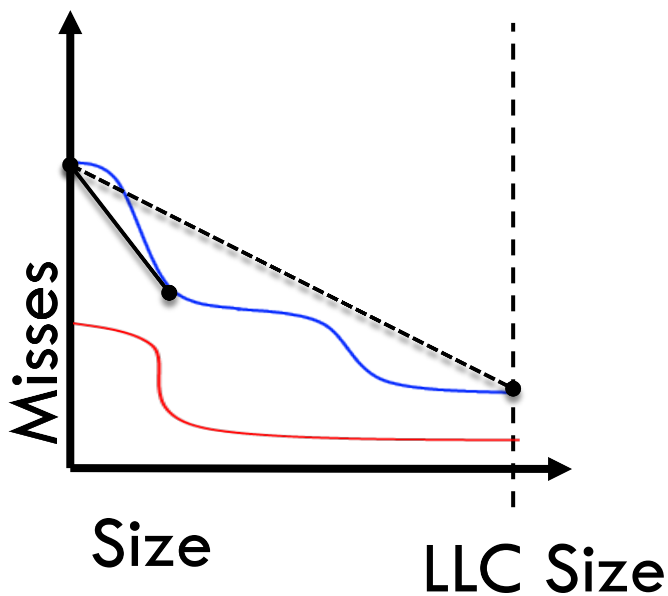
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



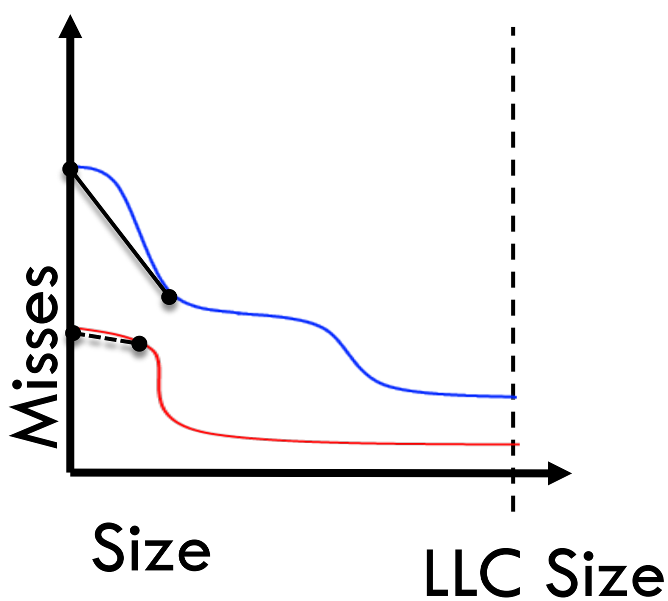
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



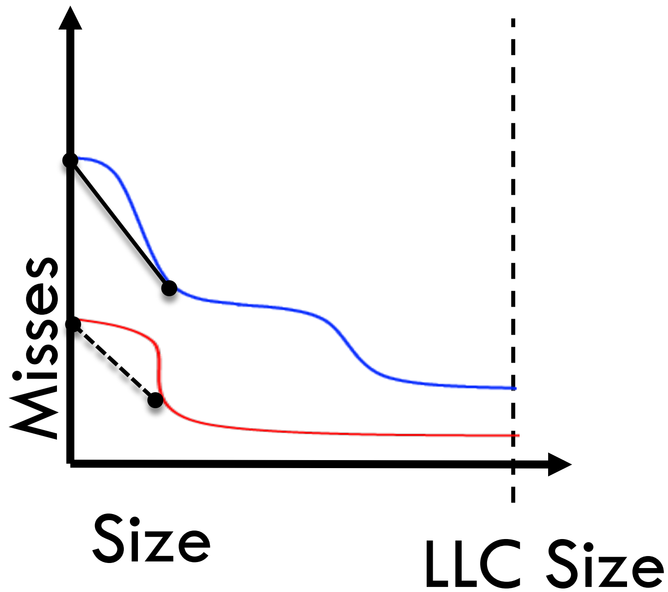
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



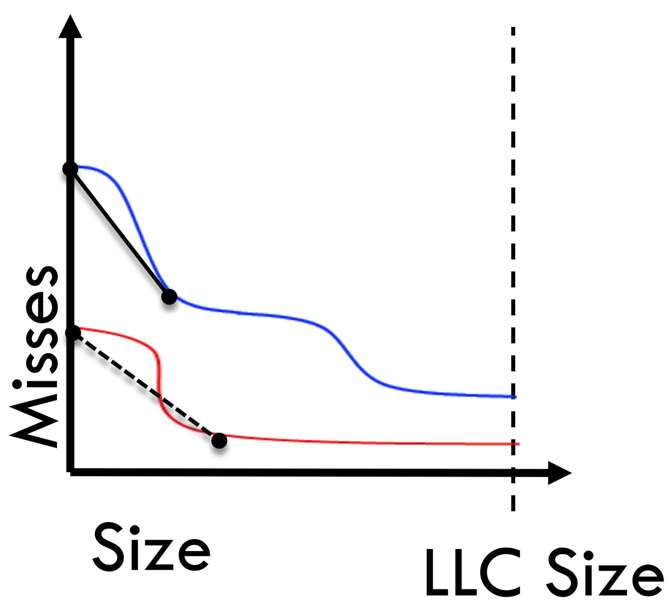
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



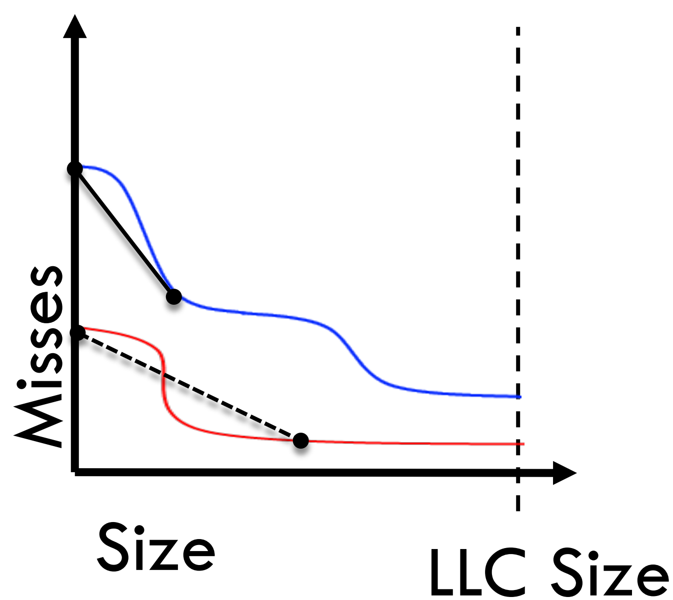
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



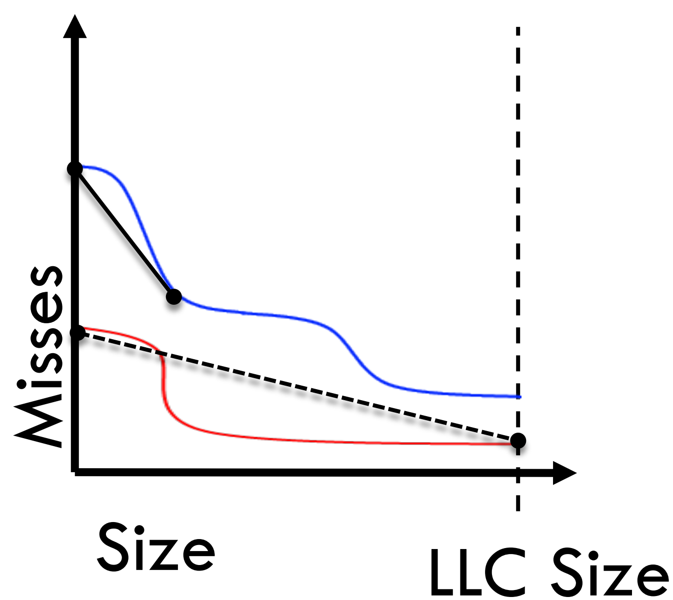
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



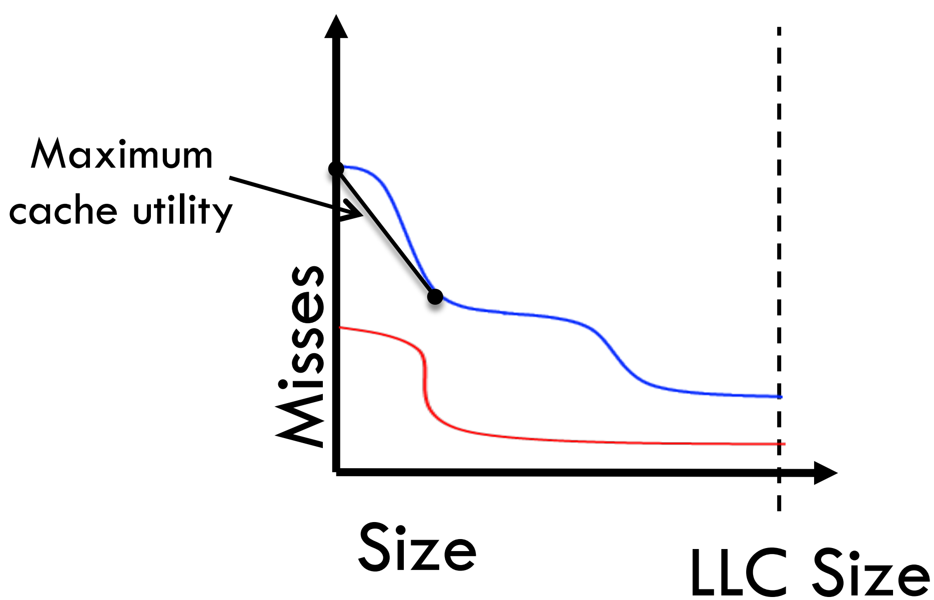
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



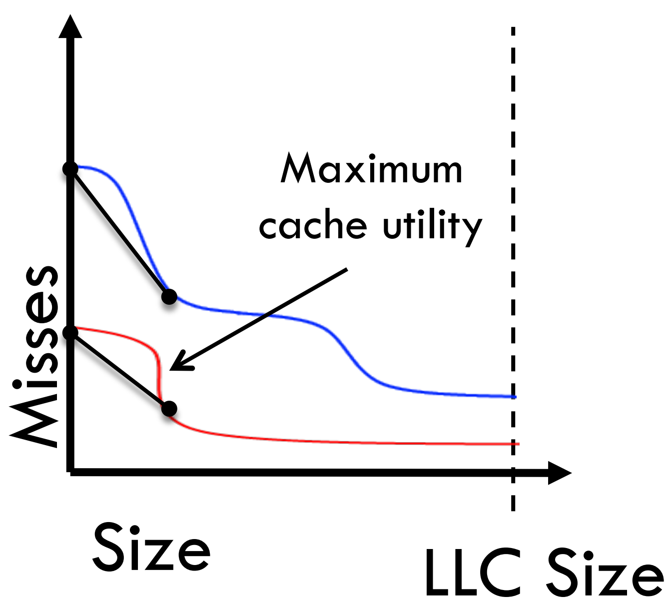
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



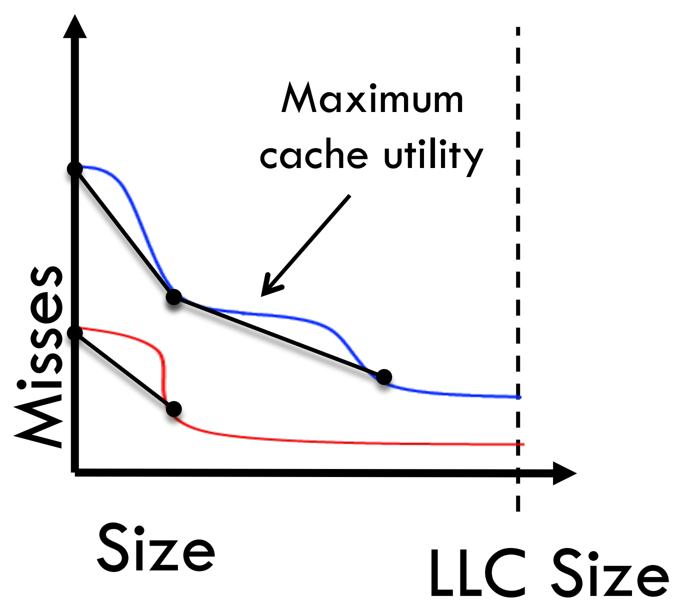
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



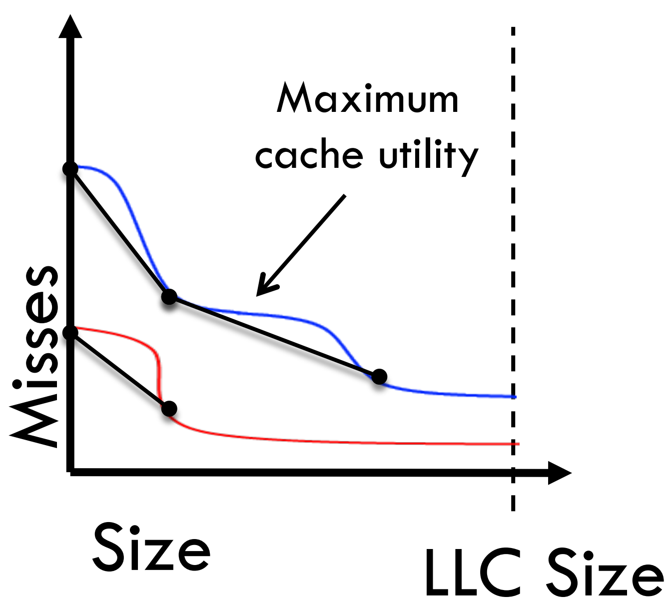
Configuration: Lookahead

- UCP Lookahead:
  - Scan miss curves to find allocation that maximizes average cache utility (hits per byte)



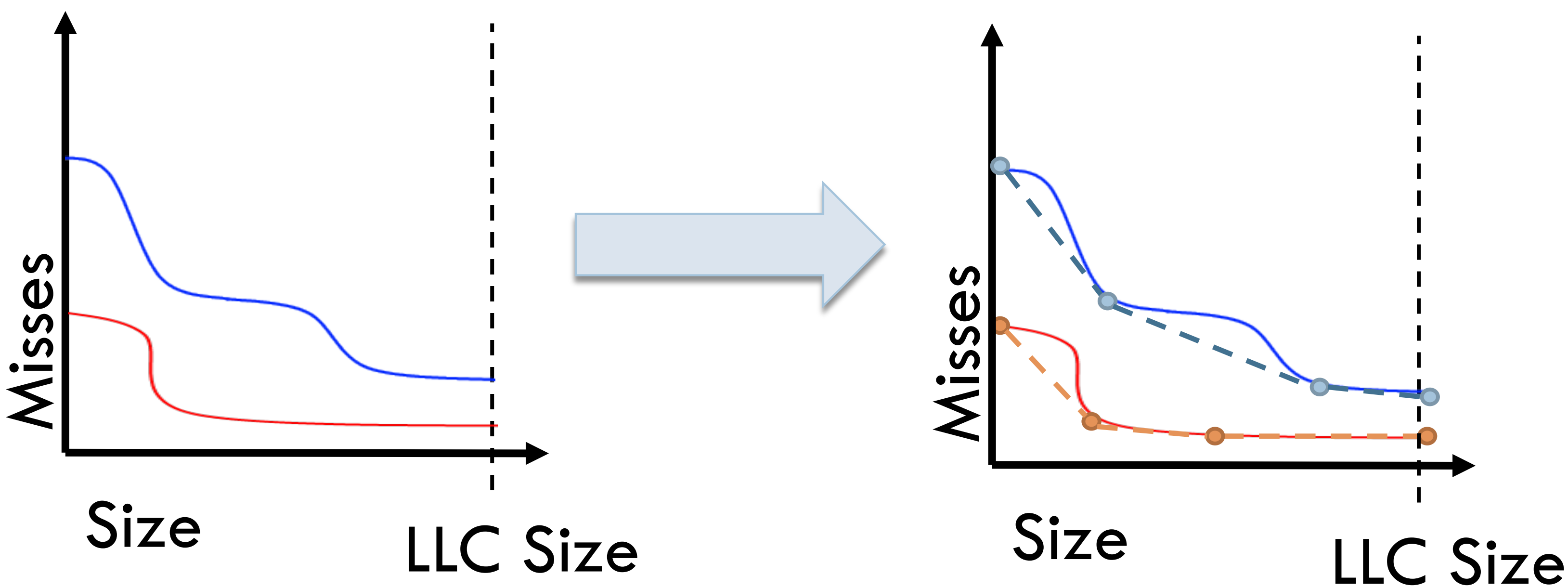
Configuration: Lookahead

- Observation: Lookahead traces the convex hull of the miss curve



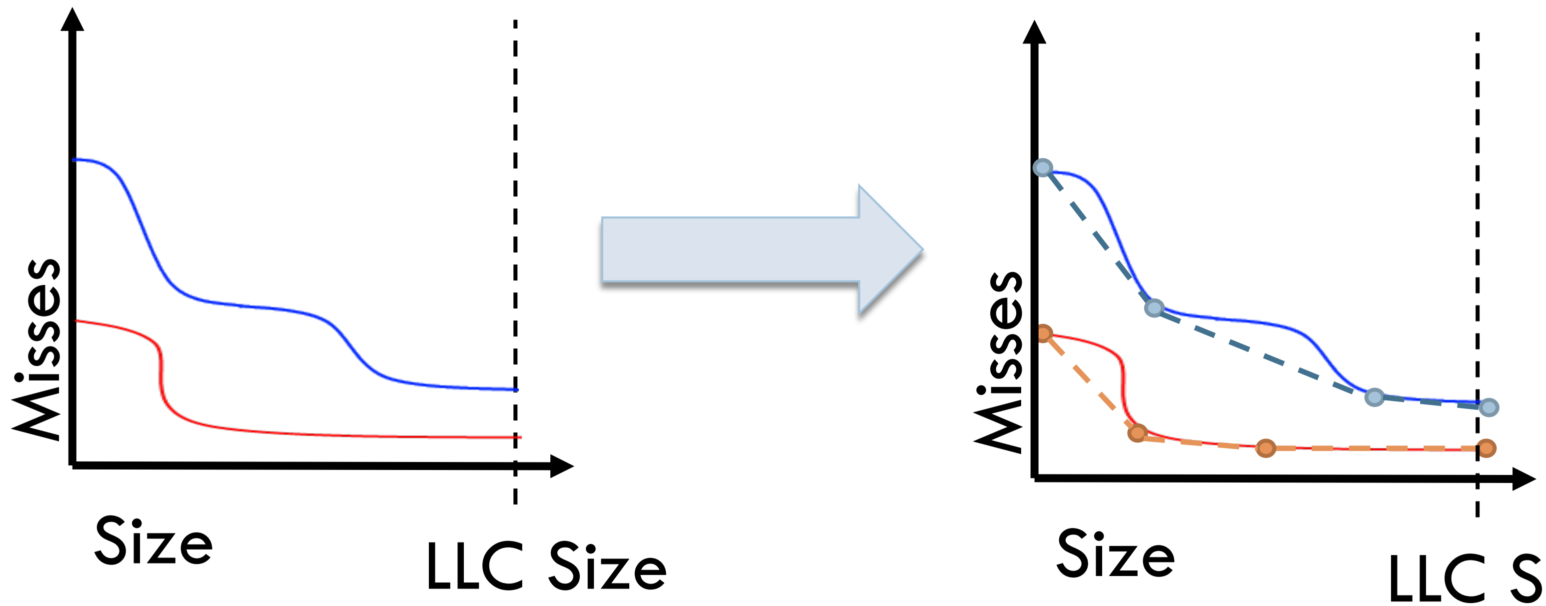
Convex Hulls

- The convex hull of a curve is the set containing all lines between any two points on the curve, or "the curve connecting the points along the bottom"



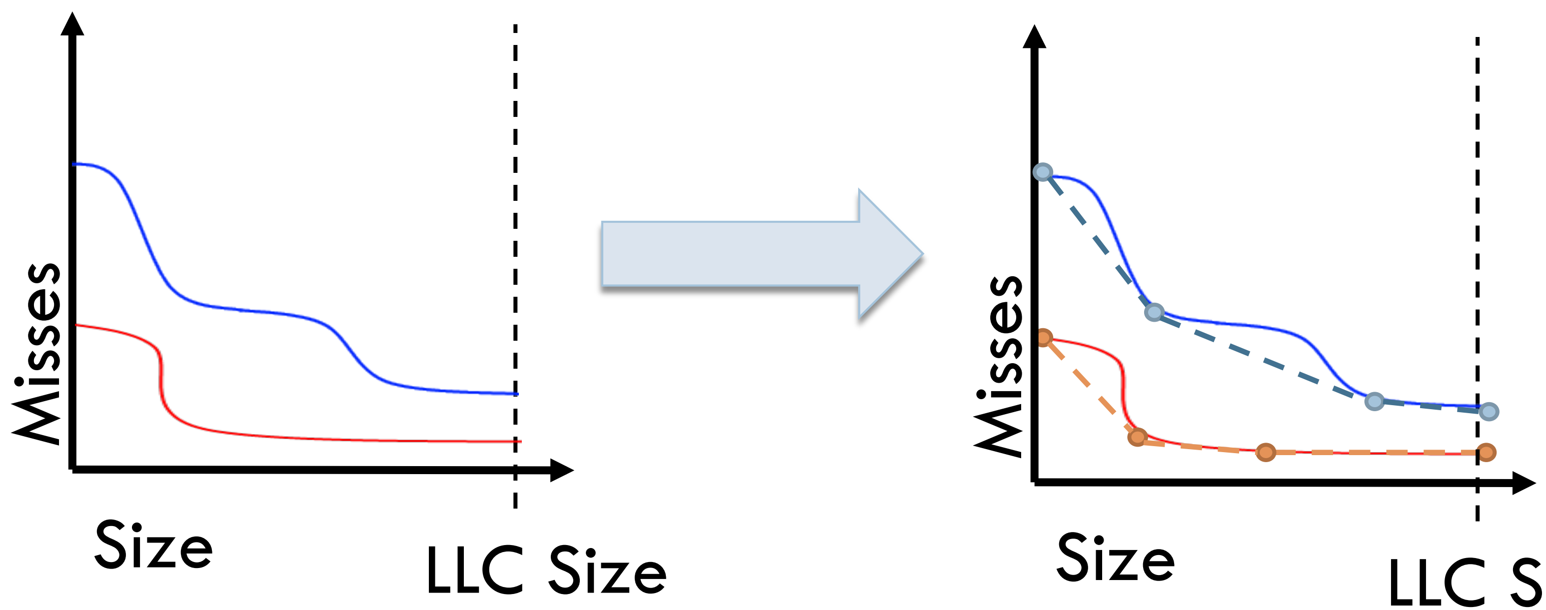
Configuration: Peekahead

- There are well-known linear algorithms to compute convex hulls
- **Peekahead algorithm is an exact, linear-time implementation of UCP Lookahead**



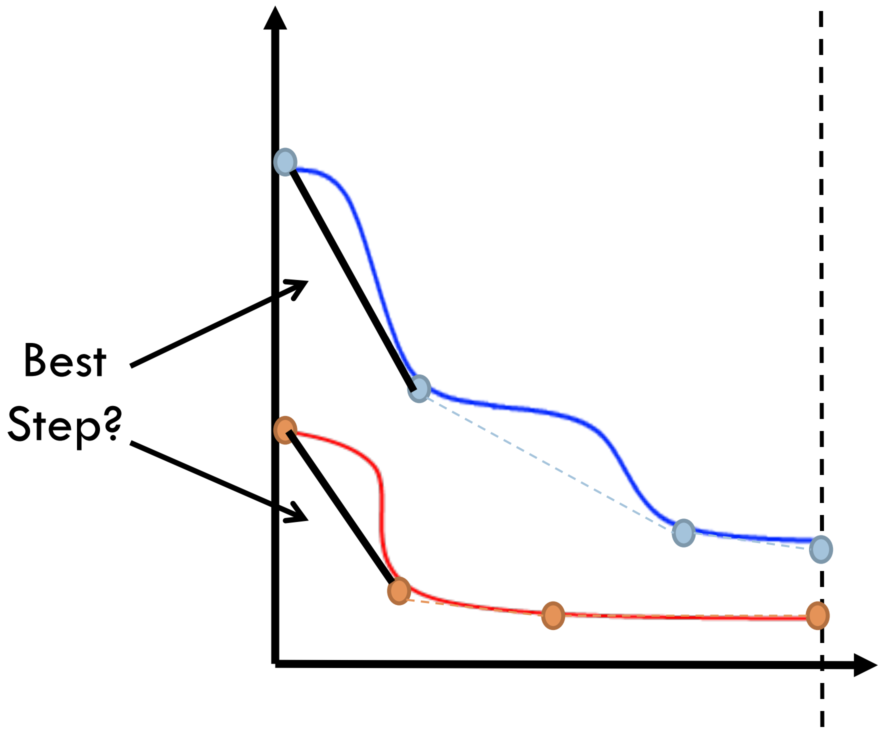
Configuration: Peekahead

- Peekahead computes all convex hulls encountered during allocation in linear time
  - Starting from every possible allocation
  - Up to any remaining cache capacity



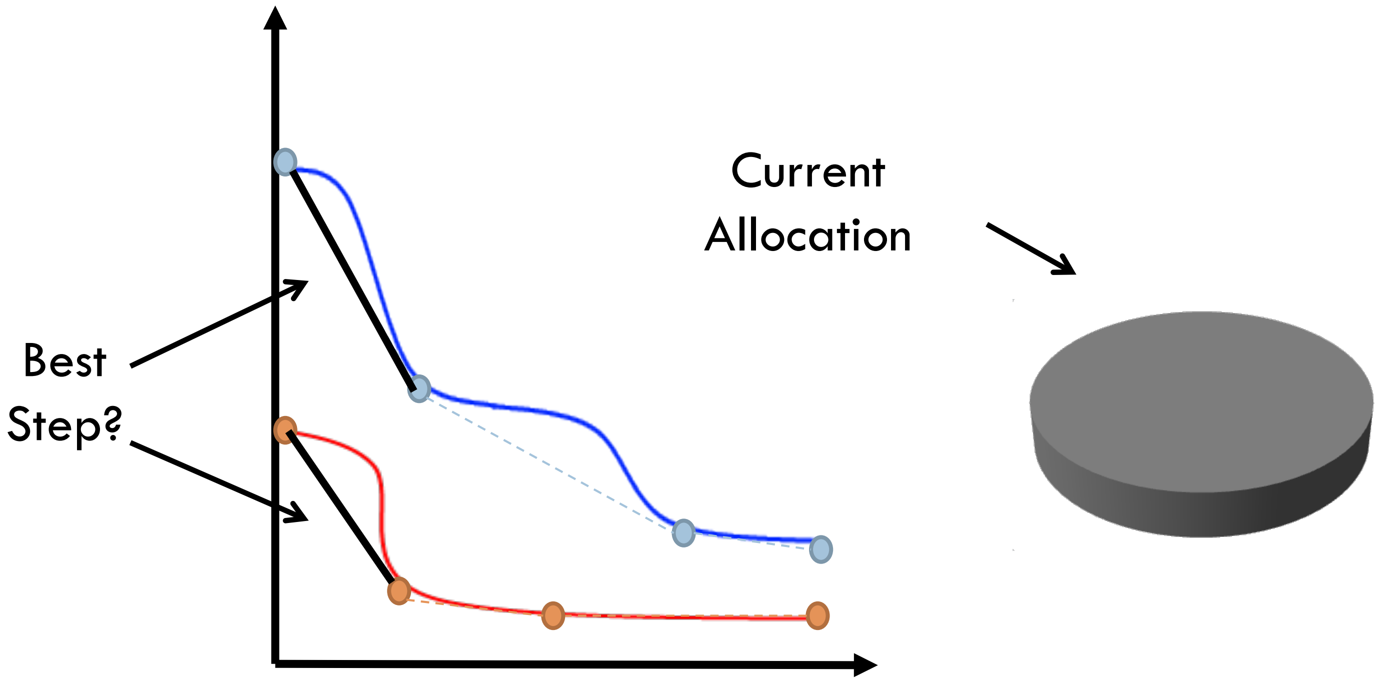
Configuration: Peekahead

- Knowing the convex hull, each allocation step is O(log P)
  - Convex hulls have decreasing slope → decreasing average cache utility → only consider next point on hull
  - Use max-heap to compare between partitions



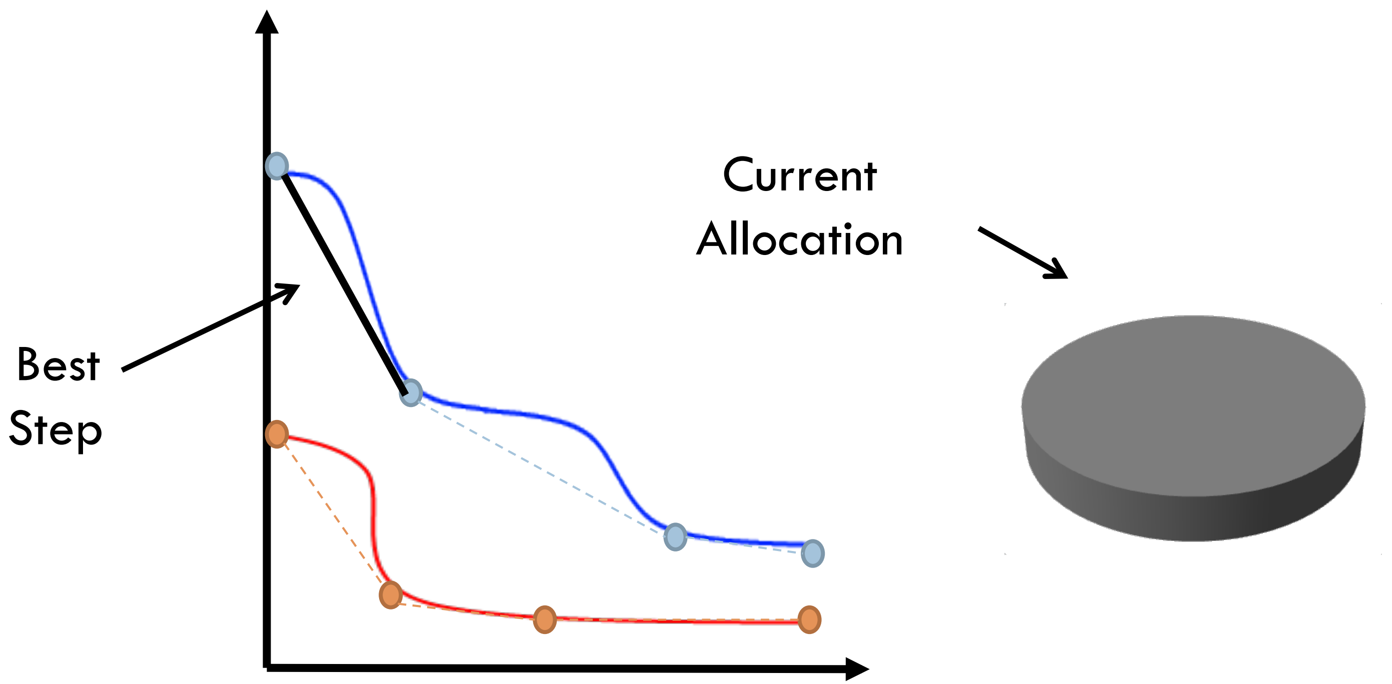
Configuration: Peekahead

- Knowing the convex hull, each allocation step is O(log P)



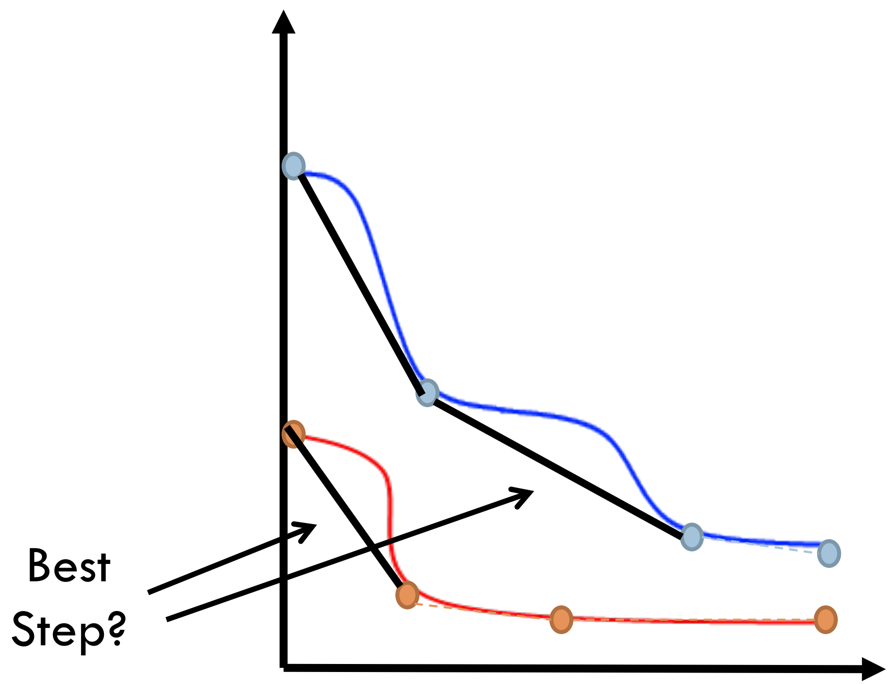
Configuration: Peekahead

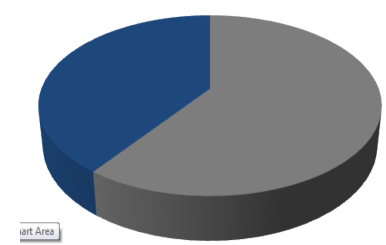
- Knowing the convex hull, each allocation step is O(log P)



Configuration: Peekahead

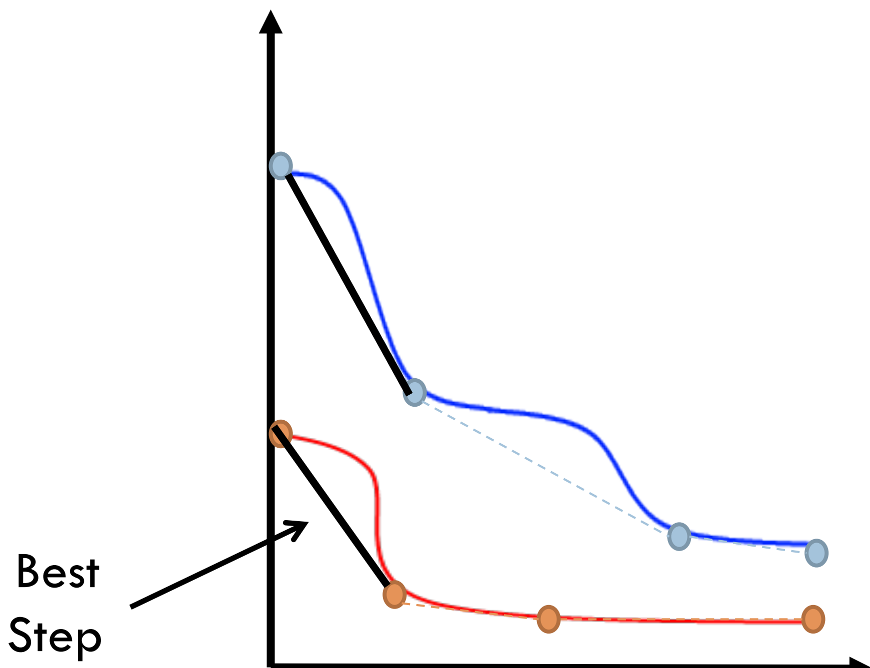
- Knowing the convex hull, each allocation step is O(log P)

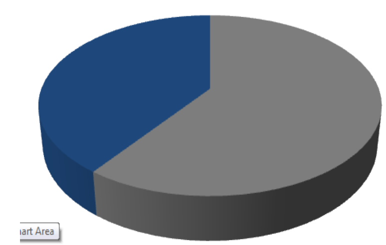




Configuration: Peekahead

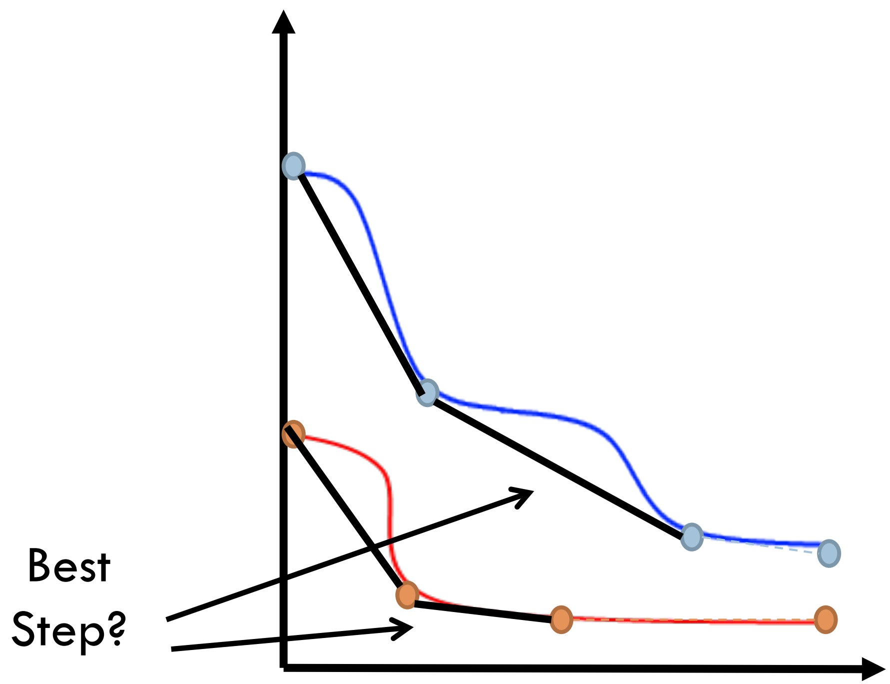
- Knowing the convex hull, each allocation step is O(log P)

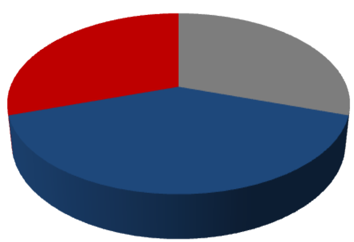




Configuration: Peekahead

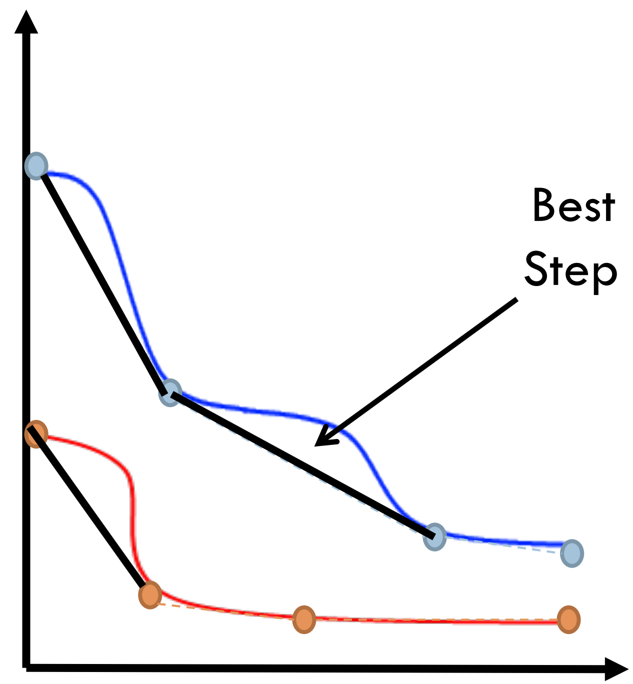
- Knowing the convex hull, each allocation step is O(log P)

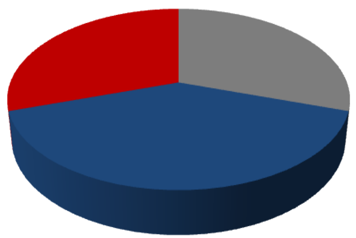




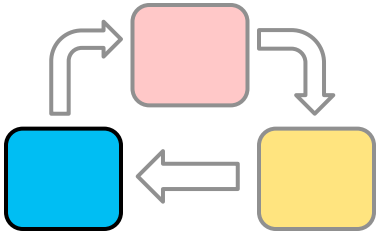
Configuration: Peekahead

- Knowing the convex hull, each allocation step is O(log P)

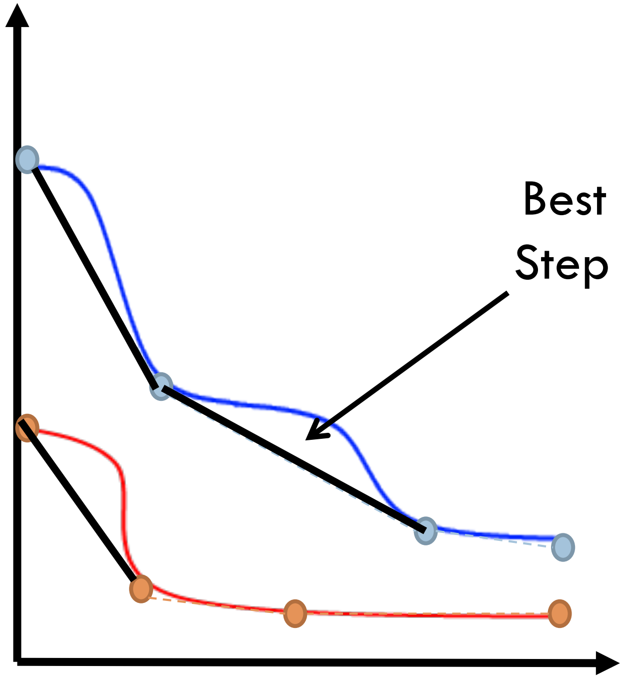


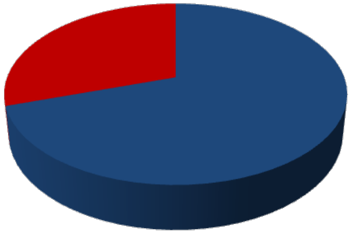


Configuration: Peekahead



- Knowing the convex hull, each allocation step is O(log P)





Configuration: Peekahead

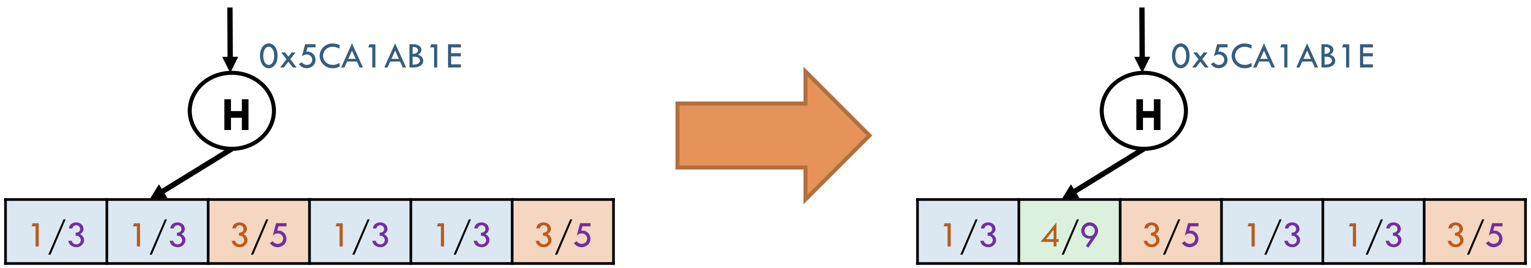
- Full runtime is O(P x S)
  - P – number of partitions
  - S – cache size

- See paper for additional examples, algorithm, and corner cases

- See technical report for additional detail, proofs, and run-time analysis
  - **Jigsaw: Scalable Software-Defined Caches (Extended Version)**, Nathan Beckmann and Daniel Sanchez, *Technical Report MIT-CSAIL-TR-2013-017, Massachusetts Institute of Technology, July 2013*

Re-configuration

- When STB changes, some addresses hash to different banks



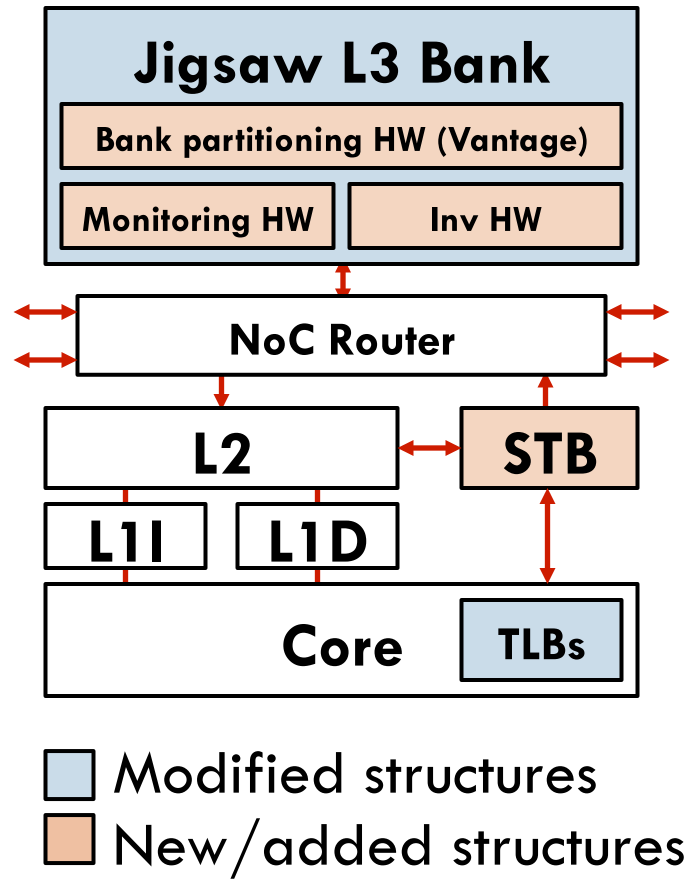
- Selective invalidation hardware walks the LLC and invalidates lines that have moved

- Heavy-handed but infrequent and avoids directory
  - **Maximum** of 300K cycles / 50M cycles = 0.6% overhead

Design: Hardware Summary

- Operation:
  - Share-bank translation buffer (STB) handles accesses
  - TLB augmented with share id
- Monitoring HW: produces miss curves
- Configuration: invalidation HW
- Partitioning HW (Vantage)

Tile Organization



Agenda

- Introduction
- Background
- Jigsaw Design
- Evaluation
  - Methodology
  - Performance
  - Energy

Methodology

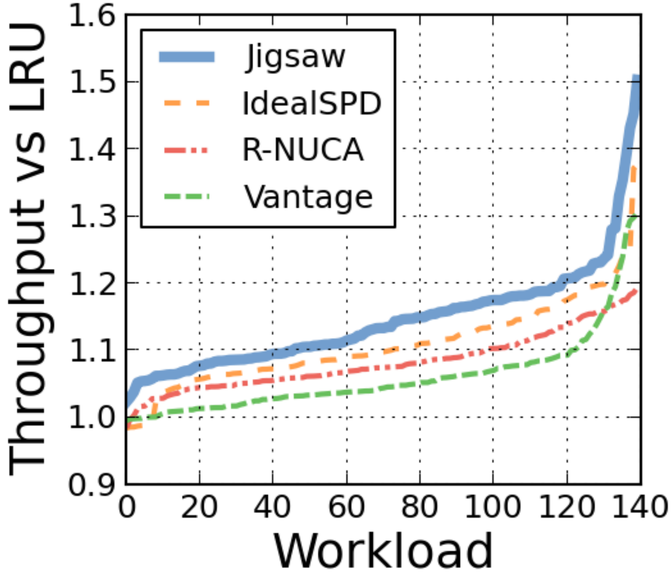
- Execution-driven simulation using zsim

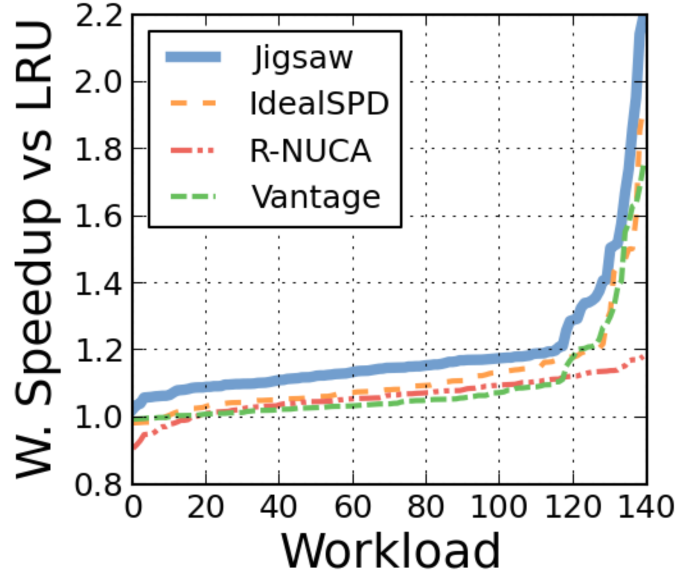
- Workloads:
  - 16-core **singlethreaded** mixes of SPECCPU2006 workloads
  - 64-core **multithreaded** (4x16-thread) mixes of PARSEC

- Cache organizations
  - LRU – shared S-NUCA cache with LRU replacement; baseline
  - Vantage – S-NUCA with Vantage and UCP Lookahead
  - R-NUCA – state-of-the-art shared-baseline D-NUCA organization
  - IdealSPD ("shared-private D-NUCA") – private L3 + shared L4
    - 2x capacity of other schemes
    - *Upper bound for private-baseline D-NUCA organizations*
  - Jigsaw

Evaluation: Performance

- 16-core multiprogrammed mixes of SPECCPU2006

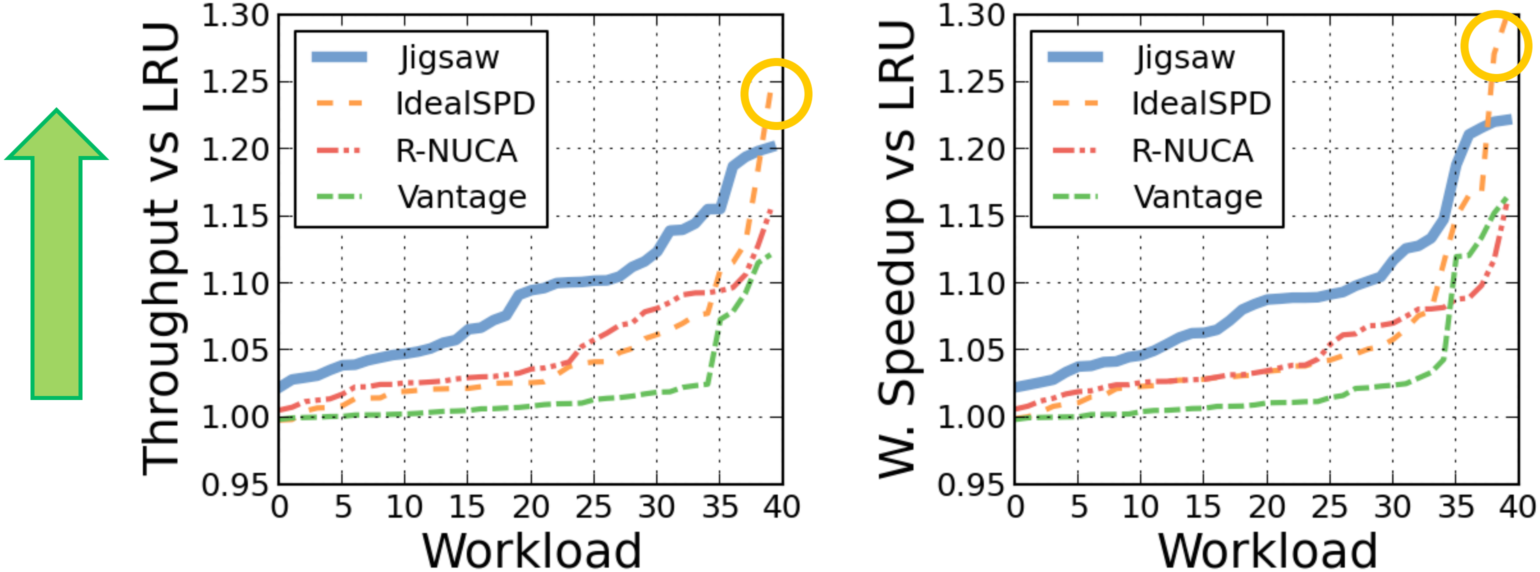




- Jigsaw achieves best performance
  - Up to 50% improved throughput, 2.2x improved w. speedup
  - Gmean +14% throughput, +18% w. speedup
- Jigsaw does even better on the most memory intensive mixes
  - Top 20% of LRU MPKI
  - Gmean +21% throughput, +29% w. speedup

Evaluation: Performance

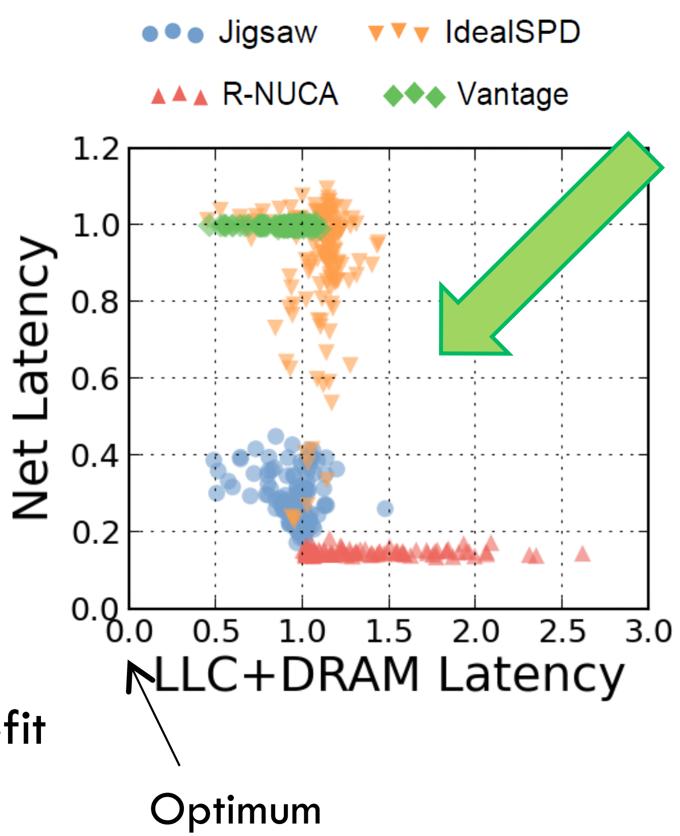
- 64-core multithreaded mixes of PARSEC



- Jigsaw achieves best performance
  - Gmean +9% throughput, +9% w. speedup
- Remember IdealSPD is an upper bound with 2x capacity

Evaluation: Performance Breakdown

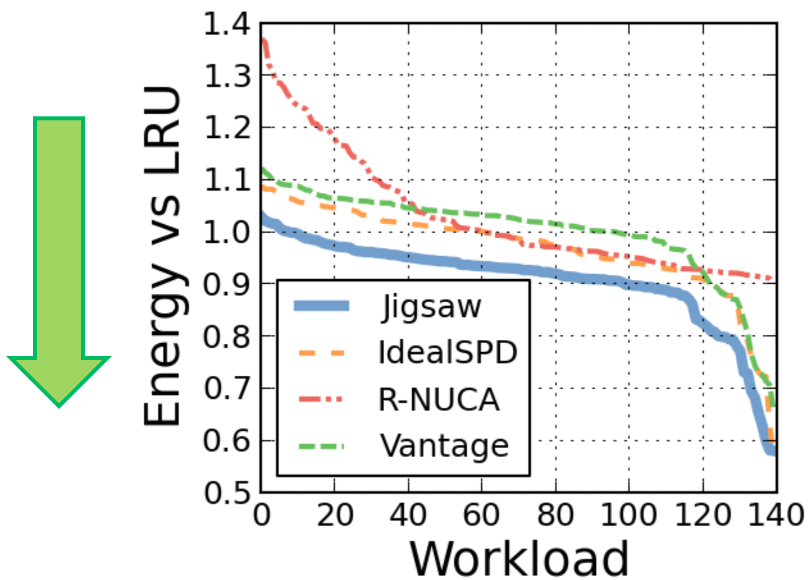
- 16-core multiprogrammed mixes of SPECCPU2006
- Breakdown memory stalls into network and DRAM
  - Normalized to LRU
- R-NUCA is limited by capacity in these workloads (private data ➔ local bank)
- Vantage only benefits DRAM
- IdealSPD acts as *either* a private organization (benefit network) *or* a shared organization (benefit DRAM)



- Jigsaw is the only scheme to *simultaneously* benefit network and DRAM latency

Evaluation: Energy

- 16-core multiprogrammed mixes

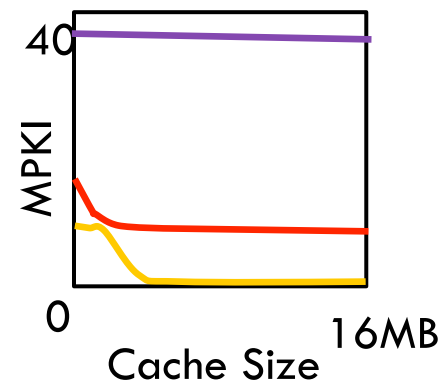


- McPAT models of full-system energy (chip + DRAM)
- Jigsaw achieves best energy reduction
  - Up to 72%, gmean of 11%
  - Reduces both network and DRAM energy

Conclusion

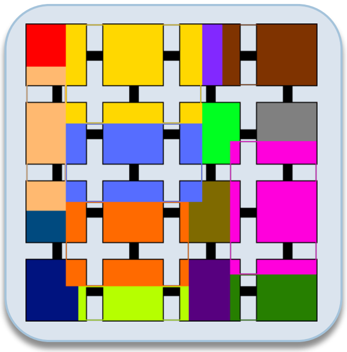
- NUCA is giving us more capacity, but further away

- Applications have widely varying cache behavior

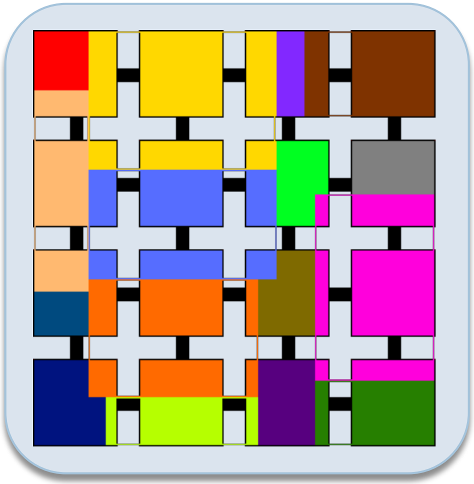


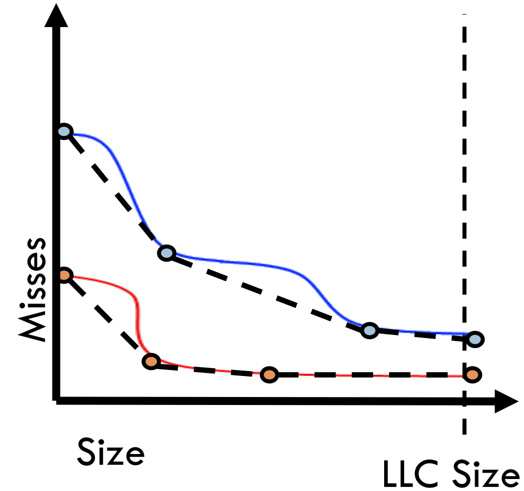
- Cache organization should adapt to meet application needs

- Jigsaw uses physical cache resources as building blocks of virtual caches, or ***shares***
  - Sized to fit working set
  - Placed near application for low latency

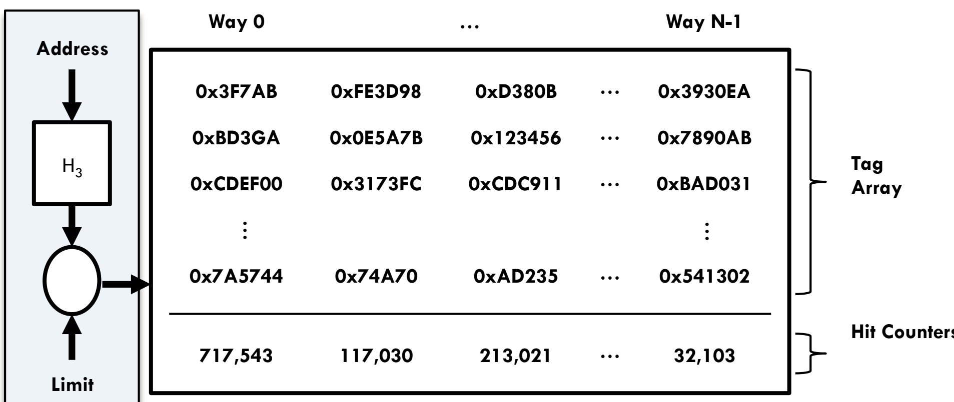


- Jigsaw improves performance up to 2.2x and reduces energy up to 72%





QUESTIONS



Massachusetts Institute of Technology

CSAIL

Placement

- Greedy algorithm
- Each share is allocated budget
- Shares take turns grabbing space in “nearby” banks
  - Banks ordered by distance from “center of mass” of cores accessing share
- Repeat until budget & banks exhausted

Monitoring

- Software requires miss curves for each share

- Add UMONs per tile
  - Small tag array that models LRU on sample of accesses
  - Tracks # hits per way, # misses → miss curve

- Changing sampling rate models a larger cache

$$\text{Sampling Rate} = \frac{\text{UMON Lines}}{\text{Modeled Cache Lines}}$$

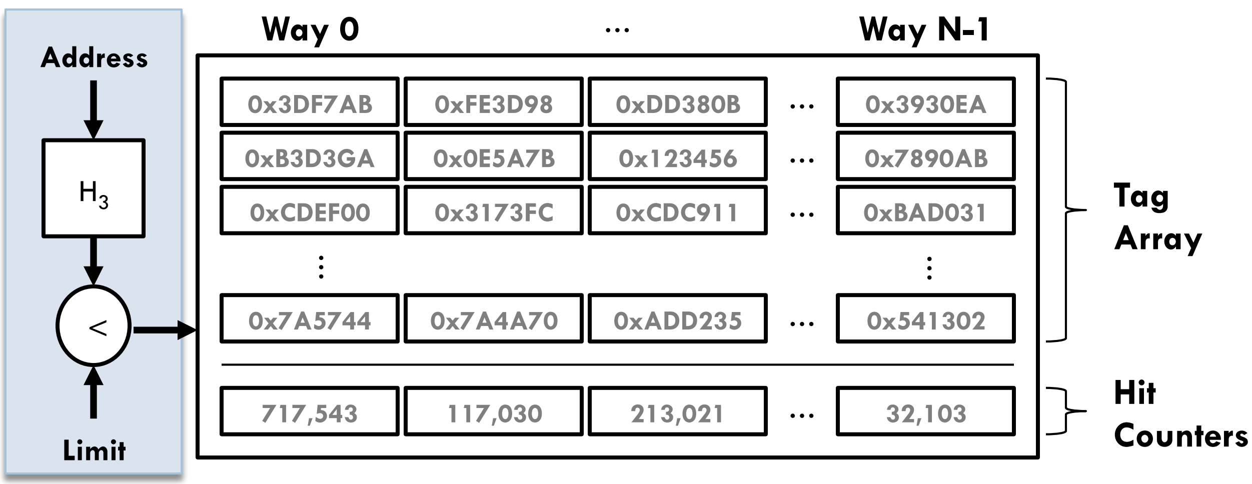
- STB spreads lines proportionally to partition size, so sampling rate must compensate

$$\text{Sampling Rate} = \frac{\text{Share size}}{\text{Partition size}} \times \frac{\text{UMON Lines}}{\text{Modeled Cache Lines}}$$

Monitoring

- STB spreads addresses unevenly → change sampling rate to compensate

- Augment UMON with hash (shared with STB) and 32-bit limit register that gives fine control over sampling rate



- UMON now models full LLC capacity exactly
  - Shares require only one UMON
  - Max four shares / bank → four UMONs / bank → 1.4% overhead

Evaluation: Extra

- See paper for:
  - Out-of-order results
  - Execution time breakdown
  - Peekahead performance
  - Sensitivity studies